ARPA ORDER NO. 2223

ISI/RR-76-49

November 1976

John V. Guttag
USC Computer Science Department

Ellis Horowitz
USC Computer Science Department

David R. Musser
USC Information Sciences Institute

The Design of Data Type Specifications

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER (14) ISI/RR-76-49	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER (9)
4. TITLE *(and Subtitle)* (6) THE DESIGN OF DATA TYPE SPECIFICATIONS.	5. TYPE OF REPORT & PERIOD COVERED Research Report.
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) John V. Guttag, USC Ellis Horowitz, USC David R. Musser, ISI	8. CONTRACT OR GRANT NUMBER(s) (15) NSF Grant No. MCS76-06089 DAHC 15-72-C-0308
9. PERFORMING ORGANIZATION NAME AND ADDRESS USC Information Sciences Institute 4676 Admiralty Way Marina del Rey, CA 90291	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order No. 2223
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209	12. REPORT DATE (11) November 1976 13. NUMBER OF PAGES 26
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)* ------ (12) 27 p.	15. SECURITY CLASS. *(of this report)* UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

This document is approved for public release and sale; distribution is unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

------

18. SUPPLEMENTARY NOTES

(OVER)

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

data type, specification, algebraic axioms, software design, recursive programming, program correctness

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

(OVER)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

407952

AB

18. SUPPLEMENTARY NOTES

This report is an expanded version of a paper given at the Second International Conference on Software Engineering, October 1976. It will also appear in Current Trends in Programming Methodology, Vol. IV: Data Structuring, Raymond T. Yeh, ed. (to be published by Prentice-Hall in 1977).

20. ABSTRACT

This report concerns the design of data types in the creation of a software system; its major purpose is to explore a means for specifying a data type that is independent of its eventual implementation. The particular style of specification, called algebraic axioms, is exhibited by axiomatizing many commonly used data types. These examples reveal a great deal about the intricacies of data type specification via algebraic axioms, and also provide a standard to which alternative forms may be compared. Further uses of this specification technique are in proving the correctness of implementations and in interpretively executing a large system design before actual implementation commences.

ARPA ORDER NO. 2223

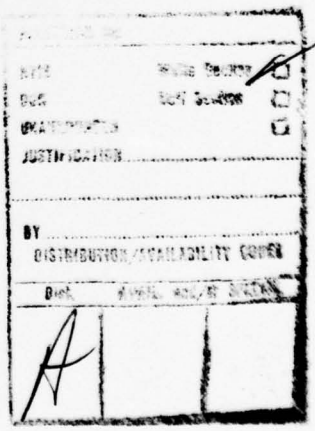
ISI/RR-76-49
November 1976

John V. Guttag
USC Computer Science Department

Ellis Horowitz
USC Computer Science Department

David R. Musser
USC Information Sciences Institute

The Design of Data Type Specifications



INFORMATION SCIENCES INSTITUTE

4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

UNIVERSITY OF SOUTHERN CALIFORNIA

This research was supported in part by the National Science Foundation under Grant No. MCS76-06089 and by the Defense Advanced Research Projects Agency under Contract No. DAHC15 72 C 0308, ARPA Order No. 2223, Program Code Nos. 3D30 and 3P10.

Views and conclusions contained in this study are the authors' and should not be interpreted as representing the official opinion or policy of ARPA, the U.S. Government or any other person or agency connected with them.

This document approved for public release and sale; distribution is unlimited.

CONTENTS

ABSTRACT

This report concerns the design of data types in the creation of a software system; its major purpose is to explore a means for specifying a data type that is independent of its eventual implementation. The particular style of specification, called algebraic axioms, is exhibited by axiomatizing many commonly used data types. These examples reveal a great deal about the intricacies of data type specification via algebraic axioms, and also provide a standard to which alternative forms may be compared. Further uses of this specification technique are in proving the correctness of implementations and in interpretively executing a large system design before actual implementation commences.

This report is an expanded version of a paper given at the Second International Conference on Software Engineering, October 1976. It will also appear in *Current Trends in Programming Methodology, Vol. IV: Data Structuring*, Raymond T. Yeh, ed. (to be published by Prentice-Hall in 1977).

1. INTRODUCTION

Creating a software system is generally regarded as a four-stage process: requirements, design, coding, and testing. For some of these stages, tools and/or techniques that significantly enhance the process have been developed. Recently, concern has increased about developing aids for the design stage. Design is essentially a creative, synthetic process, and a fully automated tool is very unlikely. What has been suggested is a "methodology" or a style of working which is purported to yield improved designs.

Top-down design is a process whereby a task is transformed into an executable program. This process in its purest form calls for carefully refining, step by step, the functional requirements of a system into operational programs. Further guidelines regarding the choice of appropriate statements and the postponement of design decisions can be found in [Dahl 72].

The purpose of this report is to explore a complementary design strategy, the design of data types. A complete software system may contain a variety of types (lists, stacks, trees, matrices, etc.) and a variety of operations. One useful design procedure is to treat those operations that act primarily on a single data type as forming a unit and to consider the semantics of these operations as the definition of the type. This idea was implicit in the SIMULA 67 programming language [Dahl 70], in which the syntactic designation *class* denotes a collection of such operations. However, the class concept applies this principle at the programming language level rather than at design time. Each operation of a class is a directly executable program. It is also useful to consider a collection of operations at design time; then the process of design (of data types) consists of specifying those operations to increasingly greater levels of detail until an executable implementation is achieved. The idea we wish to explore here is how to create an initial specification of a data type.

A *data type specification* (or abstract data type) is a representation-independent formal definition of each operation of a data type. Thus, the complete design of a single data type would proceed by first giving its specification, followed by an (efficient) implementation that agrees with the specification. This separation of data type design into two distinct phases is very useful from an organizational point of view. Any process that needs to make use of the data type can do so by examining the specification alone. There is no need to wait until the type is fully implemented, nor is it necessary to fully comprehend the implementation.

There are two chief concerns in devising a technique for data type specification. The first is to devise a notation that permits a rigorous definition of operations but remains representation-independent, and the second is to learn to use that notation. There are many criteria one can use to measure the value of a specification notation, but the two major ones are as follows:

> Can specifications be constructed without undue difficulty?
>
> Is the resulting specification easy to comprehend?

As with programming, there are potentially a very large number of ways to specify an operation. A good data type specification should give just enough information to define

the type, but not so much that the choice of implementations based upon it is limited. Thus, we say that a data type specification is an abstraction of a concept of which the eventual implementation is only one instance.

In this report our intent is to explore a particular specification technique, *algebraic specifications* [Goguen 75], [Guttag 75], [Zilles 75], by exhibiting specifications for a number of commonly used data types. Those we have chosen are typical of those that are discussed in a course on data structures; see [Horowitz 76]. By supplying these examples we hope to convince the reader that the style of specification we discuss here is especially appropriate for designing data types and that it meets the two criteria previously stated. Secondly, we hope these example specifications will provide a standard by which other methods can be compared. We do not pretend to have supplied definitive specifications of the example data types. Both our choice of operations and the semantics we associate with some of the operations are somewhat arbitrary.

In the last section we indicate how these specifications can be further used for proving the correctness of implementations and for testing, at design time, large software systems. However, since these subjects are fairly lengthy, we limit our presentation here to an informal discussion of reading and writing data type specifications. The remaining subjects will only be hinted at here, but are dealt with in [Guttag 76b].

Many other people have been working on these and related areas and we have profited from their ideas. A useful bibliography of this work is given in [Liskov 75]. Some of the particular axiomatizations have already appeared in the literature, notably Stacks, Queues, and Sets; see [Goguen 75], [Guttag 76a], [Liskov 75], [Spitzen 75], and [Standish 73].

2. *THE SPECIFICATIONS*

How can one describe a data type without unduly constraining its eventual implemented form? One method is to define the object using natural language and mathematical notation. For example, a stack can be defined as a sequence of objects $(a_1,\ldots,a_n)$ $n\geq 0$, where insertions or deletions are allowed only at the right-hand end. This type of definition is not satisfactory from a computing standpoint, where it is preferable to define constructively a data type by defining the operations which create, build up, and destroy instances of the type. Since software designers generally know how to program, the use of a programming-like language for specification is especially desirable.

The features we choose permit only the following:

1. free variables
2. *if-then-else* expressions
3. Boolean expressions
4. recursion

Moreover, we restrict the use of procedures to those which are single-valued and have no side effects. Note that many features normally presumed to be present in conventional programming languages (such as assignment to variables, iteration statements) are not permitted in this formalism. This approach may seem so arbitrary as to eliminate the

possibility of ever achieving the previously stated goals, but actually it has several strong points to recommend it. First, the restricted set yields a representation-independent means for supplying a specification. Second, the resulting specifications can clearly express the desired concepts if the reader is comfortable with reading recursive programs. (Though many programmers are not so accustomed, a faithful reading of this report will serve as a tutorial on this subject.) Third, the separation of values and side effects lends clarity and simplifies a specification. Though requiring this separation may be too restrictive for an implementation, the criterion of efficiency can be relaxed at the specification stage. Fourth, the above features can be easily axiomatized, which is a necessary first step for successfully carrying out proofs of implementations; see [Guttag 76b].

Let us begin with the very simple example of a Stack data type which is given in Figure 2.1. The operations which are available for manipulating a stack are: (1) NEWSTACK, which produces an instance of the empty stack; (2) PUSH, which inserts a new item onto the stack and returns the resulting stack; (3) POP, which removes the top item and returns the resulting stack; (4) TOP, which returns the top item of the stack; and (5) ISNEWSTACK, which tests if a stack is empty. For each operation, the types of its input and output are listed in the ***declare*** statement. Notice that all operations are true functions which return a single value and allow no side effects. If stack operations are implemented by procedures with side effects, their effect can be specified easily in terms of the operations we have given. Extending the formalism in this way is discussed in Section 4.

```
type Stack[item]
1.    declare     NEWSTACK() → Stack
2.                PUSH(Stack,item) → Stack
3.                POP(Stack) → Stack
4.                TOP(Stack) → item ∪ {UNDEFINED}
5.                ISNEWSTACK(Stack) → Boolean;
6.    for all     s ∈ Stack, i ∈ item let
7.                ISNEWSTACK(NEWSTACK) = true
8.                ISNEWSTACK(PUSH(s,i)) = false
9.                POP(NEWSTACK) = NEWSTACK
10.               POP(PUSH(s,i)) = s
11.               TOP(NEWSTACK) = UNDEFINED
12.               TOP(PUSH(s,i)) = i
13.   end
end Stack
```

Figure 2.1

At this point let us introduce the notational conventions we will use throughout this report. All operation names are written in upper case. Type names begin with a capital letter, e.g., Stack. Lower case symbols are regarded as free variables, such as s and i in Figure 2.1, which are taken to be of type Stack and item, respectively. Type names can be modified by listing "parameters" within square brackets. These parameters may be type names or free variables whose range is a type, e.g., item is such a variable and indicates that the type Stack can apply to any other data type. The equations within the ***for all*** and ***end*** are the axioms which describe the semantics of the operations.

At first these axioms may prove difficult to comprehend. One aid is to interpret the axioms as defining a set of recursive functions. The empty stack is represented by a function with no input arguments, NEWSTACK. Then asking for the topmost element of NEWSTACK is regarded as an exceptional condition which does not result in an item; hence we call it UNDEFINED. The only other stack we can have must be of the form PUSH(s,i) where s is any stack and i is the most recently inserted item. Then by line 12 the last element inserted is the first returned. Notice that we need not worry about expressions of the form TOP(POP(s)), since axioms 9 and 10 give us rules for expressing any value of type Stack in terms of only NEWSTACK and PUSH.

Unfortunately, the Stack example is far too simple in many respects to properly illustrate the intricacies of data type specification. A somewhat richer example is the data type circular list defined in Figure 2.2. This type has seven operations. Five of these, CREATE, INSERT, DELETE, VALUE, and ISEMPTY, have exact analogs in typestack. The RIGHT and JOIN operations introduce additional complexity by allowing us to rotate the list of stored elements, thus permitting access to both ends of the list, and to join two lists into one. This additional complexity is reflected in the recursion of axioms 17 and 19.

```
type   CircularList[item]
1.  declare   CREATE( ) → CircularList
2.            INSERT(CircularList, item) → CircularList
3.            DELETE(CircularList) → CircularList
4.            VALUE(CircularList) → item ∪ {UNDEFINED}
5.            ISEMPTY(CircularList) → Boolean
6.            RIGHT (CircularList) → CircularList
7.            JOIN(CircularList,CircularList) → CircularList
8.  for all   c, c1 ∈ CircularList, i, i1, i2∈ item let
9.            ISEMPTY(CREATE) = true
10.           ISEMPTY(INSERT(c,i)) = false
11.           DELETE(CREATE) = CREATE
12.           DELETE(INSERT(c,i)) = c
13.           VALUE(CREATE) = UNDEFINED
14.           VALUE(INSERT(c,i)) = i
15.           RIGHT(CREATE) = CREATE
16.           RIGHT(INSERT(CREATE,i)) = INSERT(CREATE,i)
17.           RIGHT(INSERT(INSERT(c,i),i1))
                    =INSERT(RIGHT(INSERT(c,i1)),i)
18.           JOIN(c,CREATE) = c
19.           JOIN(c,INSERT(c1,i)) = INSERT(JOIN(c,c1),i)
      end
end   CircularList
```

Figure 2.2

This specification is similar to one given by Valdis Berzins [Berzins] for a *symmetric* circular list data type, which included a LEFT operation, but not a JOIN operation.

We have now introduced almost the entire specification language used in writing algebraic axioms. All that remains is to introduce conditionals into the right-hand sides.

This is done in the definition of type Queue, a first-in first-out list, in Figure 2.3. There are six operations: four produce queues, one returns an item, and one is Boolean-valued. An easy way to understand the axioms is to conceive of the set of all queues as being represented by the set of strings consisting of

$$NEWQ \textit{ or } ADDQ(...ADDQ(ADDQ(NEWQ,i_1),i_2),...,i_n),n\geq 1.$$

The item i_1 is at the front and i_n is at the rear. Then the axioms can be concretely thought of as rules which show how each operation acts on any such string. For example, taking the FRONTQ of the empty queue is UNDEFINED. Otherwise FRONTQ is applied to a queue whose most recently inserted item is i, and q represents the remainder of the queue. If q is empty, then i is the correct result; otherwise FRONTQ is recursively applied to q. A similar situation holds for the DELETEQ operation. Notice that none of the common forms of queue representation, e.g., as linked lists or in an array, is implied or precluded by this definition.

```
type  Queue[item]
1.    declare    NEWQ( ) → Queue
2.               ADDQ(Queue,item) → Queue
3.               DELETEQ(Queue) → Queue
4.               FRONTQ(Queue) → item ∪ {UNDEFINED}
5.               ISNEWQ(Queue) → Boolean
6.               APPENDQ(Queue,Queue) → Queue;
7.    for all    q,r ∈ Queue, i ∈ item let
8.               ISNEWQ(NEWQ) = true
9.               ISNEWQ(ADDQ(q,i)) = false
10.              DELETEQ(NEWQ) = NEWQ
11.              DELETEQ(ADDQ(q,i)) =
12.                  if ISNEWQ(q) then NEWQ
13.                       else ADDQ(DELETEQ(q),i)
14.              FRONTQ(NEWQ) = UNDEFINED
15.              FRONTQ(ADDQ(q,i)) =
16.                  if ISNEWQ(q) then i else FRONTQ(q)
17.              APPENDQ(q,NEWQ) = q
18.              APPENDQ(r,ADDQ(q,i)) = ADDQ(APPENDQ(r,q),i)
19.   end
end   Queue
```

Figure 2.3

Let us consider a third familiar structure, the *binary tree* (Binarytree), and examine in more detail the virtue of regarding all values of the data structure as being represented by strings. Its specification is given in Figure 2.4.

```
type  Binarytree[item]
    declare   EMPTYTREE( ) → Binarytree
              MAKE(Binarytree,item,Binarytree) → Binarytree
              ISEMPTYTREE(Binarytree) → Boolean
              LEFT(Binarytree) → Binarytree
              DATA(Binarytree) → item ∪ {UNDEFINED}
              RIGHT(Binarytree) → Binarytree
              ISIN(Binarytree,item) → Boolean;
    for all   l,r ∈ Binarytree, d,e ∈ item let
              ISEMPTYTREE(EMPTYTREE) = true
              ISEMPTYTREE(MAKE(l,d,r)) = false
              LEFT(EMPTYTREE) = EMPTYTREE
              LEFT(MAKE(l,d,r)) = l
              DATA(EMPTYTREE) = UNDEFINED
              DATA(MAKE(l,d,r)) = d
              RIGHT(EMPTYTREE) = EMPTYTREE
              RIGHT(MAKE(l,d,r)) = r
              ISIN(EMPTYTREE,e) = false
              ISIN(MAKE(l,d,r),e) =
                   if d=e
                        then true
                        else ISIN(l,e) or ISIN(r,e)
    end
end  Binarytree
```

Figure 2.4

The operations included are EMPTYTREE, which creates the empty tree; MAKE, which joins two trees together with a new root; and operations which access the data at a node, return the left subtree or the right subtree of a node, and search for a given data item. Three operations which we might naturally wonder whether to include are the usual traversal methods (preorder, inorder, and postorder), which place the elements contained in the tree into a queue [Horowitz 76]. Perhaps the strongest reason for including them is the very fact that they are so succinctly stated by our recursive notation, e.g., INORD(Binarytree) → Queue and

```
INORD(EMPTYTREE) = NEWQ
INORD(MAKE(l,d,r)) = APPENDQ(ADDQ(INORD(l),d),INORD(r))
```

for l,r ∈ Binarytree and d ∈ item. The choice of which operations to include in a specification is arbitrary. We have omitted this operation because it makes significant use of the operations of another data type, Queue. However, this does give us the opportunity to experiment with the string representation. Let us present an example which starts with the binary tree

```
T = MAKE(MAKE(EMPTYTREE,B,EMPTYTREE),A,
                MAKE(EMPTYTREE,C,EMPTYTREE))
```

and applies the axioms to INORD(T) to obtain

```
INORD(T) = APPENDQ(ADDQ(INORD(MAKE(EMPTYTREE,B,EMPTYTREE)),A),
```

INORD(MAKE(EMPTYTREE,C,EMPTYTREE)))

which by the definition of INORD becomes

APPENDQ(ADDQ(APPENDQ(ADDQ(NEWQ,B),NEWQ),A),
APPENDQ(ADDQ(NEWQ,C),NEWQ))

and now using the axioms for APPENDQ we obtain

APPENDQ(ADDQ(ADDQ(NEWQ,B),A),ADDQ(NEWQ,C))

and again applying APPENDQ we obtain

ADDQ(APPENDQ(ADDQ(ADDQ(NEWQ,B),A),NEWQ),C)

which gives the final result

ADDQ(ADDQ(ADDQ(NEWQ,B),A),C).

At this point the reader has seen three examples, and we are in a better position to argue the virtues of the specification notation. The number of axioms is directly related to the number of operations of the type being described. The restriction of expressing axioms using only the *if-then-else* and recursion has not caused any contortions. This should not come as a surprise to LISP programmers who have found these features largely sufficient over many years of programming. One criticism we have encountered is that recursion forces one into inefficient code, as evidenced by the FRONTQ operation which finds the front element of the queue by starting at the last element. To this we reply that a specification should not be viewed as describing the eventual implemented program, but merely as a means for understanding what the operation is to do. One might also suppose that the operation names are not well chosen, and then wonder how easy it is to discern their meaning via the axioms. This is hard to respond to, especially when trying to imagine how other techniques would fare under this restriction. Nevertheless, we might ask the reader if he can determine what the operation MYSTERY does where MYSTERY(Queue) → Queue and

MYSTERY(NEWQ) = NEWQ
MYSTERY(ADDQ(q,i)) = APPENDQ(ADDQ(NEWQ,i),MYSTERY(q))

are the axioms which define it.

Let us pursue the binary tree example a bit further. In most applications the elements in the tree are somehow ordered. That is to say, the tree is built up from a series of INSERT operations that preserve some ordering relationship among the nodes of the tree. This non-primitive INSERT operation can be programmed in terms of the primitive operations of type Binarytree. One drawback of such an approach to creating a restricted kind of binary tree is that we cannot rely upon a type-checking mechanism to guarantee that the desired ordering property is always maintained. If, on the other hand, we declare a type with INSERT as a primitive operation, we can achieve the desired level of security.

Consider type Bstree (binary search tree) defined to be a binary tree with data items at each node, such that for any node its item is alphabetically greater than any item in its left subtree and alphabetically less than any item in its right subtree; see [Horowitz 76]. Some axioms have to be changed and a new operation added in order to transform the Binarytree specification into one for type Bstree. The second axiom for ISIN is altered to read

```
ISIN(MAKE(l,d,r),e) =
        if d=e then true
            else if d<e then ISIN(r,e)
                                else ISIN(i,e).
```

The new operation is INSERT(Bstree,item) → Bstree which searches for an item in a binary search tree and, if it is not there, inserts it appropriately. Note that this is the only way that a binary search tree can be created. This implies that the operation MAKE, present in the specification of type Binarytree, must not be accessible to the programmer in this new specification. If it were available, we could not guarantee that all binary search trees would be well formed. Thus we regard MAKE as a "hidden" function [Parnas 72] and attach a star to it in the new specification (Figure 2.5) to indicate that it is no longer accessible.

```
type  Bstree
      declare   EMPTYTREE( ) → Bstree
                *MAKE(Bstree,item,Bstree) → Bstree
                ISEMPTYTREE(Bstree) → Boolean
                LEFT(Bstree) → Bstree
                DATA(Bstree) → item ∪ {UNDEFINED}
                RIGHT(Bstree) → Bstree
                ISIN(Bstree,item) → Boolean,
                INSERT(Bstree;,item) → Bstree;
      for all   l,r ∈ Bstree, d,e ∈ item let
                ISEMPTYTREE(EMPTYTREE) = true
                ISEMPTYTREE(MAKE(l,d,r)) = false
                LEFT(EMPTYTREE) = EMPTYTREE
                LEFT(MAKE(l,d,r)) = l
                DATA(EMPTYTREE) = UNDEFINED
                DATA(MAKE(l,d,r)) = d
                RIGHT(EMPTYTREE) = EMPTYTREE
                RIGHT(MAKE(l,d,r)) = r
                ISIN(EMPTYTREE,e) = false
                ISIN(MAKE(l,d,r),e) =
                      if d=e then true
                           else if d<e then ISIN(r,e) else ISIN(l,e)
                INSERT(EMPTYTREE,e) = MAKE(EMPTYTREE,e,EMPTYTREE)
                INSERT(MAKE(l,d,r),e) =
                      if d=e then MAKE(l,d,r)
                           else if d<e then MAKE(l,d,INSERT(r,e))
                                       else MAKE(INSERT(l,e),d,r)
      end
end   Bstree
```

Figure 2.5

Let us consider another familiar type, String. In the specification of Figure 2.6, we have chosen five primitive operations: NULL, which creates the null string; ADDCHAR, which appends a character to a string; CONCAT, which joins two strings together; SUBSTR(s,i,j), which from a string s returns the j-character substring beginning at the i^{th} character of s; INDEX(s,t), which returns the position of the first occurrence of a string t as a substring of a string s (0 if t is not a substring of s).

```
type String
  declare NULL( ) → String
          ISNULL(String) → Boolean
          LEN(String) → Integer
          ADDCHAR(String,Character) → String
          CONCAT(String,String) → String
          SUBSTR(String,Integer,Integer) → String
          INDEX(String,String) → Integer;
  for all s,t ∈ String, c,d ∈ Character, i,j ∈ Integer let
          ISNULL(NULL) = true
          ISNULL(ADDCHAR(s,c)) = false
          LEN(NULL) = 0
          LEN(ADDCHAR(s,c)) = LEN(s)+1
          CONCAT(s,NULL) = s
          CONCAT(s,ADDCHAR(t,d)) = ADDCHAR(CONCAT(s,t),d)
          SUBSTR(NULL,i,j) = NULL
          SUBSTR(ADDCHAR(s,c),i,j) =
                if j = 0
                   then NULL
                   else if j = LEN(s)-i+2
                              then ADDCHAR(SUBSTR(s,i,j-1),c)
                              else SUBSTR(s,i,j)
          INDEX(s,NULL) = LEN(s)+1
          INDEX(NULL,ADDCHAR(t,d)) = 0
          INDEX(ADDCHAR(s,c),ADDCHAR(t,d)) =
                if INDEX(s,ADDCHAR(t,d)) ≠ 0
                   then INDEX(s,ADDCHAR(t,d))
                   else if c=d and INDEX(s,t) = LEN(s)-LEN(t)+1
                              then INDEX(s,t)
                              else 0
  end
end String
```

Figure 2.6

Notice that there are several types which make up this definition in addition to type String: namely, types Character, Integer, and Boolean. In general, a data type specification always defines only one type, but it may require the operations of other data types to accomplish this. Another question which arises again is when should an operation be part of the specification and when should it not, an issue we have already encountered with binary trees. The operations we have chosen here are basically those provided in PL/1.

So far we have concentrated primarily on how to read axioms. Now let us consider how to create them. As a general outline of attack we begin with a basic set of operations $f_1,...,f_m$. A subset of these, say $f_1,...,f_k$ $k \leq m$, have as their output the data type being defined. Out of the k operations are chosen a subset which we call the *constructor set*, satisfying the property that all instances of the data type can be represented using only constructor set operations. Then the axioms which need to be written are those that show how each non-constructor-set operation behaves on all possible instances of the data type.

As a new example consider the type Set. The operations whose range is of type Set are: EMPTYSET, which has the usual meaning; INSERT and DELSET, which put an element into or delete one from the set, respectively. Out of these three operations we select EMPTYSET and INSERT as the constructors. Then an arbitrary set containing $n \geq 1$ items is given by the expression

$$\text{INSERT}(...\text{INSERT}(\text{EMPTYSET},i_1),...,i_n).$$

A very important feature of this definition is the fact that there is no ordering assumed on the items. Alternatively, the specification might insist that $i_1 < i_2 < ... < i_n$ also be true.

```
type Set[item]
  declare   EMPTYSET( ) → Set
            ISEMPTYSET(Set) → Boolean
            INSERT(Set,item) → Set
            DELSET(Set,item) → Set
            HAS(Set,item) → Boolean;
  for all   s ∈ Set, i,j ∈ item let
            ISEMPTYSET(EMPTYSET) = true
            ISEMPTYSET(INSERT(s,i)) = false
            HAS(EMPTYSET,i) = false
            HAS(INSERT(s,i),j) =
               if i=j then true else HAS(s,j)
            DELSET(EMPTYSET,i) = EMPTYSET
            DELSET(INSERT(s,i),j) =
               if i=j then DELSET(s,j)
                  else INSERT(DELSET(s,j),i)
  end
end Set
```

Figure 2.7

The next example, the Graph type in Figure 2.8, is interesting in several respects. The mathematical definition of a graph is generally in terms of two sets: nodes and edges. This is reflected in the constructors for this definition which are EMPTYGRAPH, ADDNODE, and ADDEDGE. This definition allows for an unconnected graph and for nodes with no edges incident to them. An edge is given by the function REL(i,j) (a constructor of the data type Edge), and it is not specified whether the edges are directed or not. Notice that three of the operations result in sets, and the parameter notation has been naturally

extended to distinguish between sets with different types of elements. ADJAC finds all nodes which are adjacent to some vertex. NODOUT(g,v) removes the node v and all edges incident to v. EDGEOUT removes a single edge from the graph.

```
type Graph
  declare   EMPTYGRAPH( ) → Graph
            ADDNODE(Graph,Node) → Graph
            ADDEDGE(Graph,Edge) → Graph
            NODES(Graph) → Set(Node)
            EDGES(Graph) → Set(Edge)
            ADJAC(Graph,Node) → Set(Node)
            NODOUT(Graph,Node) → Graph
            EDGEOUT(Graph,Edge) → Graph;
  for all   g ∈ Graph, i,j,k,l,v,w ∈ Node let
            NODES(EMPTYGRAPH) = EMPTYSET
            NODES(ADDNODE(g,v)) = INSERT(NODES(g),v)
            NODES(ADDEDGE(g,REL(i,j))) = INSERT(INSERT(NODES(g),i),j)
            EDGES(EMPTYGRAPH) = EMPTYSET
            EDGES(ADDNODE(g,v)) = EDGES(g)
            EDGES(ADDEDGE(g,REL(i,j))) = INSERT(EDGES(g),REL(i,j))
            ADJAC(EMPTYGRAPH,v) = EMPTYSET
            ADJAC(ADDNODE(g,w),v) = ADJAC(g,v)
            ADJAC(ADDEDGE(g,REL(i,j)),v) =
                if v=i then INSERT(ADJAC(g,v),j)
                    else if v=j then INSERT(ADJAC(g,v),i)
                        else ADJAC(g,v)
            NODOUT(EMPTYGRAPH,v) = EMPTYGRAPH
            NODOUT(ADDNODE(g,w),v) =
                if v=w then NODOUT(g,v) else ADDNODE(NODOUT(g,v),w)
            NODOUT(ADDEDGE(g,REL(i,j)),v) =
                if v=i or v=j then NODOUT(g,v)
                    else ADDEDGE(NODOUT(g,v),REL(i,j))
            EDGEOUT(EMPTYGRAPH,REL(i,j)) = EMPTYGRAPH
            EDGEOUT(ADDNODE(g,v),REL(i,j)) =
                ADDNODE(EDGEOUT(g,REL(i,j)),v)
            EDGEOUT(ADDEDGE(g,REL(k,l)),REL(i,j)) =
                if REL(k,l) = REL(i,j) then g
                    else ADDEDGE(EDGEOUT(g,REL(i,j)),REL(k,l))
  end
end Graph
```

Figure 2.8

The next example is a sequential File data type (Figure 2.9). The operations include READ, WRITE, RESET, ISEOF (end-of-file check), and SKIP (past a specified number of records).

```
type File[record]
  declare   EMPTYFILE( ) → File
            WRITE(File,record) → File
            SKIP(File,Integer) → File
            RESET(File) → File
            ISEOF(File) → Boolean
            READ(File) → record ∪ {UNDEFINED}
  for all   f ∈ File, r,s ∈ record, i,j ∈ Integer let
            SKIP(EMPTYFILE,i) = EMPTYFILE
            SKIP(SKIP(f,j),i) = SKIP(f,j+i)
            RESET(EMPTYFILE) = EMPTYFILE
            RESET(WRITE(f,r)) = SKIP(WRITE(f,r),0)
            RESET(SKIP(WRITE(f,r),i)) = SKIP(WRITE(f,r),0)
            ISEOF(EMPTYFILE) = true
            ISEOF(WRITE(f,r)) = true
            ISEOF(SKIP(WRITE(f,r),i)) =
                  if i=0 then false else ISEOF(SKIP(f,i-1))
            READ(EMPTYFILE) = UNDEFINED,
            READ(WRITE(f,r)) = UNDEFINED,
            READ(SKIP(WRITE(f,r),i)) =
                  if ISEOF(SKIP(f,i))
                        then r
                        else READ(SKIP(f,i))
            WRITE(SKIP(WRITE(f,r),i),s)
                  if ISEOF(SKIP(f,i))
                        then WRITE(f,s)
                        else WRITE(SKIP(f,i),s)
  end
end File
```

Figure 2.9

Sequential file operations would not, in practice, be implemented as functions, but rather as procedures with side effects, say READP(f,r) and WRITEP(f,r). The operations we have given can be used to specify the effects of these procedures: READP(f,r) means r ← READ(f), f ← SKIP(f,1); and WRITEP(f,r) means f ← WRITE(f,r). Note that the axioms imply that if a SKIP operation immediately follows a WRITE, it means reset the file to its beginning, then skip past i records. Also, if a record is overwritten, the part of the file past that record is lost. For further study of the axioms note that all File values can be viewed as one of the following string forms:

EMPTYFILE *or* WRITE(WRITE(...(EMPTYFILE,r_1),...),r_n)
or SKIP(WRITE(WRITE(...(EMPTYFILE,r_1),...),r_n),i).

We conclude this section with a presentation of type Polynomial. The usual mathematical definition of a polynomial is an expression of the form

$$a_m x^m + a_{m-1} x^{m-1} + \ldots + a_1 x + a_0$$

where x is an indeterminate and the a_i's come from some commutative ring. If $a_m \neq 0$, then

m is called the degree, a_m the leading coefficient, and $a_{m-1}x^{m-1}+...+a_1x+a_0$ the reductum. A specification of Polynomials as a data type with eleven operations in given in Figure 2.10.

```
type Polynomial
  declare   ZERO( ) → Polynomial
            ADDTERM(Polynomial,Coef,Exp) → Polynomial
            REMTERM(Polynomial,Exp) → Polynomial
            MULTTERM(Polynomial,Coef,Exp) → Polynomial
            ADD(Polynomial,Polynomial) → Polynomial
            MULT(Polynomial,Polynomial) → Polynomial
            REDUCTUM(Polynomial) → Polynomial
            ISZERO(Polynomial) → Boolean
            COEF(Polynomial,Exp) → Coef
            DEGREE(Polynomial) → Exp
            LDCF(Polynomial) → Coef;
  for all   p,q ∈ Polynomial, c,d ∈ Coef, e,f ∈ Exp let
            REMTERM(ZERO,f) = ZERO
            REMTERM(ADDTERM(p,c,e),f) =
                  if e=f then REMTERM(p,f)
                        else ADDTERM(REMTERM(p,f),c,e)
            MULTTERM(ZERO,d,f) = ZERO
            MULTTERM(ADDTERM(p,c,e),d,f) =
                  ADDTERM(MULTTERM(p,d,f),c*d,e+f)
            ADD(p,ZERO) = p
            ADD(p,ADDTERM(q,d,f)) = ADDTERM(ADD(p,q),d,f)
            MULT(p,ZERO) = ZERO
            MULT(p,ADDTERM(q,d,f)) = ADD(MULT(p,q),MULTTERM(p,d,f))
            REDUCTUM(p) = REMTERM(p,DEGREE(p))
            ISZERO(ZERO) = true
            ISZERO(ADDTERM(p,c,e)) =
                  if COEF(p,e) = -c
                        then ISZERO(REMTERM(p,e))
                        else false
            COEF(ZERO,e) = 0
            COEF(ADDTERM(p,c,e),f) =
                  if e=f then c+COEF(p,f) else COEF(p,f)
            DEGREE(ZERO) = 0
            DEGREE(ADDTERM(p,c,e)) =
                  if e> DEGREE(p)
                        then e
                        else if e< DEGREE(p)
                              then DEGREE(p)
                              else if COEF(p,e) = -c
                                    then DEGREE(REDUCTUM(p))
                                    else DEGREE(p)
             LDCF(p) = COEF(p,DEGREE(p))
  end
end Polynomial
```

Figure 2.10

In this specification every Polynomial is either ZERO or constructed by applying ADDTERM to a Polynomial. Note the absence of assumptions about order of exponents, non-zero coefficients, etc., which are important as representation decisions but not essential for the specification.

The real virtue of this specification is that a fairly complex object has been completely defined using only a few lines. The corresponding programs in a conventional programming language may be several times this size. (This will be especially true if some of the "fast" algorithms are used.)

3. *CORRECTNESS OF IMPLEMENTATIONS*

Algebraic specifications of data types can play a significant role in program verification. As with any axiomatic approach, they permit factorization of proofs into distinct, manageable stages; also, the use of pure functions and equations as the form of specification permits proofs to be constructed in large part as sequences of substitutions using the equations as rewrite rules. These points are developed at length in [Guttag76b], which also describes a "data type verification system" (implemented in INTERLISP) capable of interactively assisting a human user in carrying through many of the steps of verifications automatically. In this report we shall confine our discussion of verification issues to a brief example of the implementation of one data type, Queue (Figure 2.3), in terms of another, Circular Lists (Figure 2.2). We first give, in a notation very similar to that for the specifications, an implementation of the Queue type consisting of a "representation" declaration and a "program" for each of the Queue operations in terms of the representation.

```
implementation  QueueByCircularList[item]
  declare   QREP(CircularList) → Queue
  for all   c,c1∈ CircularList, i∈ item let
            NEWQ = QREP(CREATE)
            ADDQ(QREP(c),i) = QREP(RIGHT(INSERT(c,i)))
            DELETEQ(QREP(c)) = QREP(DELETE(c))
            FRONTQ(QREP(c)) = VALUE(c)
            ISNEWQ(QREP(c)) = ISEMPTY(c)
            APPENDQ(QREP(c),QREP(c1)) = QREP(JOIN(c1,c))
  end
end   QueueByCircularList
```

Figure 3.1

A proof of correctness of this implementation consists of showing that each of the Queue axioms of Figure 2.3 is satisfied. For some of the axioms this is quite trivial because of the close correspondence between the axiomatizations of some of the Queue and CircularList operations. For example, we show that the Queue axiom ISNEWQ(NEWQ) = *true* is satisfied by the following sequence of steps:

(ISNEWQ(NEWQ) = *true*)

=[by NEWQ program]=> (ISNEWQ(QREP(CREATE)) = *true*)

=[by ISNEWQ program]=> (ISEMPTY(CREATE) = *true*)

=[by ISEMPTY axiom]=> (*true* = *true*)

=[by equality axiom]=> *true*.

A more difficult case is the following APPENDQ axiom:

[APPENDQ(q,ADDQ(r,i)) = ADDQ(APPENDQ(q,r),i)]

=[by substitution of q=QREP(c), r=QREP(c1)]=>

[APPENDQ(QREP(c),ADDQ(QREP(c1),i)) = ADDQ(APPENDQ(QREP(c),QREP(c1)),i)]

=[by ADDQ and APPENDQ programs]=>

[APPENDQ(QREP(c),QREP(RIGHT(INSERT(c1,i)))) = ADDQ(QREP(JOIN(c1,c)),i)]

=[by APPENDQ and ADDQ programs]=>

[QREP(JOIN(RIGHT(INSERT(c1,i)),c)) = QREP(RIGHT(INSERT(JOIN(c1,c),i)))].

The proof can now be completed by using the following theorem about the JOIN operation.

Theorem. JOIN(RIGHT(INSERT(c1,i)),c2) = RIGHT(INSERT(JOIN(c1,c2),i))

This theorem will be proved from the CircularList axioms using "data type induction," i.e., induction on the number of operations of the data type which are performed to obtain an element of the type (called "generator induction" in [Spitzen75]). Proofs by data type induction are often simplified if one first proves a "normal form lemma" for the data type, which specifies a minimal set of constructors of the data type (cf. the discussion of constructors following the String data type example in Section 2). For circular lists we have the following:

Normal Form Lemma: For every c∈ CircularList, (c=CREATE) or (∃c'∈ CircularList, i'∈ item such that c=INSERT(c',i'))

Proof: By data type induction. Let c be a circular list, then one of the following cases holds:

1. c = CREATE
2. c = INSERT(c1,i1)
3. c = DELETE(c1)
4. c = RIGHT(c1)

for some c1,i1. In cases 1 and 2, the theorem is clearly satisfied. In case 3, we use the induction hypothesis to conclude that c1=CREATE or ∃c2,i2 such that c1=INSERT(c2,i2). If c1=CREATE, then c=DELETE(CREATE)=CREATE, by a DELETE axiom. Otherwise, c=DELETE(INSERT(c2,i2))=c2, by the other DELETE axiom. The induction hypothesis applies

to c2, so c=c2=INSERT(c3,i3) for some c3 and i3. A similar argument proves the lemma for case 4.

Proof of Theorem. By data type induction. By the lemma, it is sufficient to consider the cases

1. c2 = CREATE
2. c2 = INSERT(c3,i3) for some c3,i3.

In case 1 we have

[JOIN(RIGHT(INSERT(c1,i)),CREATE) = RIGHT(INSERT(JOIN(c1,CREATE),i))]

=[by JOIN axiom]=> [RIGHT(INSERT(c1,i))=RIGHT(INSERT(c1,i))]

=>***true.***

In case 2 we have

[JOIN(RIGHT(INSERT(c1,i)),INSERT(c3,i3)) = RIGHT(INSERT(JOIN(c1,INSERT(c3,i3)),i))]

=[by JOIN axiom]=>

[INSERT(JOIN(RIGHT(INSERT(c1,i)),c3),i3) = RIGHT(INSERT(INSERT(JOIN(c1,c3),i3),i))]

=[by RIGHT axiom]=>

[INSERT(JOIN(RIGHT(INSERT(c1,i)),c3),i3) = INSERT(RIGHT(INSERT(JOIN(c1,c3),i)),i3)]

=[by induction hypothesis]=>

[INSERT(RIGHT(INSERT(JOIN(c1,c3),i)),i3) = INSERT(RIGHT(INSERT(JOIN(c1,c3),i)),i3)]

=> ***true.***

Thus the theorem has been proved and the APPENDQ axiom has been shown to be satisfied. Many other useful theorems (or "invariants") about data types can be proved from the axioms using the same techniques of case analysis and induction as in the foregoing proofs. In some cases these techniques can also be applied to prove theorems about an implementation. We used one such "implementation invariant" in the proof of the APPENDQ axiom without explicitly mentioning it, namely (∃c such that q=QREP(c)). This is easily proved from the Normal Form Lemma, the programs for CREATE and INSERT, and data type induction.

The proofs of the other Queue axioms for the circular list implementation require no additional techniques and will be omitted. All of these proofs have been carried through semiautomatically by the "data type verification system" described more fully in [Guttag76b].

4. PROCEDURES AND BOUNDED TYPES

Until now all of the abstract data types that we have axiomatized have been unbounded. It is relevant to observe a parallel here between computer science and mathematics, i.e., that bounded types are often harder to define than unbounded ones. In this section we intend to deal with the added complications of specifying more realistic data types, in particular a type of bounded size. At the same time we will relax the restriction that all operations be single-valued and permit a notation that resembles the conventional use of procedures, first introduced in [Guttag 76b].

It will now be permissible to include procedures in the specifications. A procedure P whose first argument, x, is altered as a result of its execution, but not its second argument, y, is syntactically declared as P(*var* x,y). If P is a pure procedure, i.e., it returns no value, then this is syntactically expressed by writing P(*var* x,y) → . The definition of procedure P would be included in the semantic specification of the data type using it. A procedure has a body and an optional value part separated by a semicolon, e.g.,

$$P(var\ x,\ var\ y) = x \leftarrow F(x,y),\ y \leftarrow G(x);\ H(x,y)$$

is a possible definition of P where F,G,H are functions returning a value. Notice that simultaneous assignment to parameters is now permitted, but we continue to adhere to our earlier approach by requiring that the value returned by a procedure be expressed by single-valued functions. In some cases these latter operations will no longer be accessible by the user of the data type. We call them "hidden functions" and indicate them by placing a star next to their names.

As an example, we give in Figure 4.1 the specification of a queue of bounded size. Notice that in comparison with the unbounded queue of Figure 2.3, four new operations have been added. ADDQ and DELETEQ are now designated as hidden functions and in their place the user will apply the pure procedure ENQ and the function DEQ, both of which have the side effect of altering their first argument. SIZE returns the number of elements contained in a bounded queue, and LIMIT the maximum number of elements permitted. Notice also that we have augmented the UNDEFINED operation by allowing it to be qualified. This will facilitate the handling of errors by distinguishing their source.

This technique of taking a specification of an unbounded data type and refining it into a bounded one can be applied in exactly the same way to yield specifications for bounded stacks, binary trees, strings, etc.

```
type Bqueue[item]
  declare   NEWQ(Integer) → Bqueue
            *ADDQ(Bqueue,item) → Bqueue
            *DELETEQ(Bqueue) → Bqueue
            FRONTQ(Bqueue) → item ∪ {UNDEFINED}
            ISNEWQ(Bqueue) → Boolean
            APPENDQ(Bqueue,Bqueue) → Bqueue
            SIZE(Bqueue) → Integer
            LIMIT(Bqueue) → Integer,
            ENQ(var Bqueue,item) → ,
            DEQ(var Bqueue) → item;
  for all   q,r ∈ Bqueue  i ∈ item let
            ISNEWQ(NEWQ(in)) = true
            ISNEWQ(ADDQ(q,i)) = false
            DELETEQ(NEWQ(in)) = NEWQ
            DELETEQ(ADDQ(q,i)) =
               if ISNEWQ(q) then NEWQ
                           else ADDQ(DELETEQ(q),i)
            FRONT(NEWQ(in)) = UNDEFINED[underflow]
            FRONT(ADDQ(q,i)) =
               if ISNEWQ(q) then i else FRONTQ(q)
            APPENDQ(q,NEWQ(in)) = q
            APPENDQ(r,ADDQ(q,i)) = ADDQ(APPENDQ(r,q),i)
            LIMIT(NEWQ(in))  = in
            LIMIT(ADDQ(q,i)) = LIMIT(q)
            ENQ(q,i) = if SIZE(q)<LIMIT(q)
                              then q ← ADDQ(q,i)
                              else q ← UNDEFINED[overflow]
            DEQ(q) = q ← DELETEQ(q); FRONTQ(q)
            SIZE(NEWQ(in)) = 0
            SIZE(ADDQ(q,i)) = 1+SIZE(q)
  end
end Bqueue
```

Figure 4.1

5. *OTHER DIRECTIONS*

In this paper we have stressed the art of data type specification. Our major goal has been to explore a notation which is especially attractive for formally defining a data type without regard to its implementation. In this section we want to indicate briefly how these specifications can be used to design reliable software, but to reserve a complete discussion for [Guttag 76b].

The first use of an axiomatic specification is as an aid in designing and implementing the type. A decision is made to choose a particular form of implementation. This implementation will be in terms of other data types and we assume that their specifications already exist. For a complex data type this process may proceed through several levels before an executable implementation is achieved. The virtue of the specifications is that each stage is made clearer by organizing the types, values, and operations that can be used.

A second use of these specifications, and perhaps its most important, is for proving that an implementation is correct. Establishing correctness now becomes equivalent to showing that the original axioms are satisfied by the newly developed implementation. This process also lends itself quite readily to automation.

Another use of these specifications is for early testing. It would be very desirable if one could design a system in such a way that it could be tested before committing people to actually build it. Given suitable restrictions on the form that the axiomatic equations may take, a system can be constructed in which implementations and algebraic specifications of data types are interchangeable. In the absence of an implementation, the operations of the data type may be interpreted symbolically. Thus, except for a significant loss in efficiency, the lack of an implementation can be made completely transparent to the user. Interestingly, it is not necessary to spend many man-years developing this system; the capability is essentially available in LISP-based symbol manipulation systems such as SCRATCHPAD [Griesmer 71], REDUCE [Hearn 71], and MACSYMA [Martin 71]. The use of REDUCE for this purpose is discussed in [Guttag 76b], as are the essential ideas of a pattern-match compiler designed especially for compilation of algebraic axioms.

ACKNOWLEDGMENTS

This report is an expanded version of a paper given at the Second International Conference on Software Engineering, October 1976. A number of people made valuable comments on earlier drafts of this paper, including Ed Lazowska, Ralph London, Mary Shaw, Tim Standish, Ron Tugender, Dave Wile, and the referees of the conference version. We are grateful to Betty Randall for her expertise and patience in typing many drafts of the paper.

REFERENCES

[Berzins] V. Berzins, "Behavioral Specifications as a Design Aid," MIT Position Paper, Version I, undated.

[Dahl 70] O. J. Dahl, B. Myhrhaug, and K. Nygaard, "The SIMULA 67 common base language," Norwegian Computing Center, Oslo, Publication S-22, 1970.

[Dahl 72] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, 1972, pp. 1-82.

[Goguen 75] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright, "Abstract data-types as initial algebras and correctness of data representations," *Proceedings, Conference on Computer Graphics, Pattern Recognition and Data Structure*, May 1975.

[Griesmer 71] J. H. Griesmer and R. D. Jenks, "SCRATCHPAD/1--An interactive facility for symbolic mathematics," *Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation*, ACM, March 1971, pp. 42-58.

[Guttag 75] J. V. Guttag, "The specification and application to programming of abstract data types," Ph. D. Thesis, University of Toronto, Department of Computer Science, 1975, available as Computer Systems Research Group Report CSRG-59.

[Guttag 76a] J. V. Guttag, "Abstract data types and the development of data structures," Supplement to the *Proceedings of the SIGPLAN/SIGMOD Conference on Data: Abstraction, Definition, and Structure*, March 1976, pp. 37-46.

[Guttag 76b] J. V. Guttag, E. Horowitz, and D. R. Musser, *Abstract Data Types and Software Validation*, USC/Information Sciences Institute, RR-76/48, August 1976.

[Hearn 71] A. C. Hearn, "Reduce 2: a system and language for algebraic manipulation," *Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation*, ACM, March 1971, pp. 128-133.

[Horowitz 76] E. Horowitz and S. Sahni, *Fundamentals of Data Structures*, Computer Science Press, Woodland Hills, California, June 1976.

[Liskov 75] B. H. Liskov and S. N. Zilles, "Specification techniques for data abstractions," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 1, March 1975, pp. 7-18.

[Martin 71] W. A. Martin and R. J. Fateman, "The MACSYMA system," *Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation*, ACM, March 1971, pp. 59-75.

[McCarthy 63] J. McCarthy, "Basis for a mathematical theory of computation," *Computer Programming and Formal Systems*, P. Braffort and D. Hirchberg (eds.), North-Holland Publishing Company, 1963, pp. 33-70.

[Parnas 72] D. L. Parnas, "A technique for the specification of software modules with examples," *Communications of the ACM*, Vol. 15, pp. 330-336, May 1972.

[Spitzen 75] J. Spitzen and B. Wegbreit, "The verification and synthesis of data structures," *Acta Informatica*, 4, 127-144 (1975).

[Standish 73] T. A. Standish, "Data structures: an axiomatic approach," Bolt Beranek and Newman Report 2639, 1973.

[Zilles 75] S. N. Zilles, "Algebraic specification of data types," Massachusetts Institute of Technology, Project MAC Progress Report 11, 1975.

DISTRIBUTION LIST

Defense Advanced Research Projects Agency
1400 Wilson Blvd., Arlington, VA 22209

Defense Documentation Center
Cameron Station
Alexandria, VA 22314

Steven Swart,ACO
Office of Naval Research
1030 Green Street
Pasadena, CA 91106